# (NeXT Tip #36) printf conversion specifiers

Christopher Lane (*lane[at]CAMIS.Stanford.EDU*)
*Tue, 29 Jun 1993 17:54:43 -0700 (PDT)*

For most programs, you rarely use more than the standard %d, %s & %f format conversion characters in 'printf'. Starting with NeXTSTEP 3.0, the 'C' compiler now checks the arguments to 'printf' calls against the conversion specifiers in the format string. To clear your programs of compiler warnings, you may need to start using format conversion characters you've not used before. A quick summary of the 'printf' format conversion specifications:

%{flags}{{field width}.{precision}}{data width}<conversion>

flags = 0 -- pad with zeros instead of spaces
- -- left justify
+ -- always print signed numbers (including positive ones)
-- <space> leave space for a number sign even if none needed
# -- alternate form, see the 'man' page for this one

field width = digit string specifying output width (padded as needed)

precision = the number of digits to appear after the decimal point

data width = h -- argument is a short integer (d/i, o, x/X, or u)
l -- argument is a long integer (d/i, o, x/X, or u)

conversion = c -- a single character
d or i -- signed decimal notation
e -- float/double printed as '[-]m.ddde+/-xx' (scientific)
f -- float/double printed as '[-]mmm.ddd'
g -- float/double printed in style e or f based on number
n -- the number of characters converted (see below)
o -- unsigned octal notation
p -- a (void *) pointer (implementation dependent)
s -- a string (char *)
u -- unsigned integer printed in decimal notation
x or X -- hexadecimal notation (capital letters if `X')
% -- just prints a '%'

The 'data width' modifier (h or l) and unsigned conversion type (u) are what you'll typically need to clear up compiler warnings you run into under NeXTSTEP 3.0 -- e.g. use '%lu' for printing a long unsigned integer instead of just '%d' which worked previously without warning messages.

The next level of sophistication when it comes to 'printf' conversion formats is the use of field width and precision. These can be used with integers and strings, as well as floating point numbers. (Their meaning varies according to which conversion type they're used with and you should consult the 'printf' manual page for details.) Below are example 'printf' calls extracted from our 'Examples' directories that use field width and precision:

```
printf("%1d", windowType);
printf("%.0f seconds", size / 1024);
printf("%5.3f %5.3f %5.3f", r, g, b);
printf("%8.3f\n", angle * 10.0 / TWOPI);
printf("%.2d/%.2d/%.4d", month, day, year);
printf("%s/Library/MolViewer/el%02d.tiff", getenv("HOME"), i);
printf("Alt:%6.2f Az:%6.2f", theta * 57.295787, chi * 57.295787);

printf("%.*s/%s", dirlen, p, file);
```

The last example is a rare but powerful usage in which the field width and/or precision may be '*' instead of a digit string. In this case an integer argument to the 'printf' call supplies the field width or precision. Here's a silly example (output & program) using dynamic (*) field width and precision:

```
#define SIZE 10
0
000 void main()
00000 {
0000000 int i, width = SIZE, minimum = -1;
000000000
00000000000 for(i = 0; i < SIZE; i++)
0000000000000 (void) printf("%*.*d\n", width++, minimum += 2, 0);
000000000000000
00000000000000000 exit(EXIT_SUCCESS);
0000000000000000000 }
```

Another underutilized feature of 'printf' is the '%n -- number of characters' conversion. According to the 'man' page, "The integer location pointed to by the int pointer arg is written the number of characters converted to this point by this call to 'printf'. No argument is converted.". In other words, '%n' takes an integer pointer argument in which it will put the number of characters output by the 'printf' call up to the point that '%n' appeared.

Here's a simple program using '%n' that reads in a file and prints it back out with a hexadecimal number in front of each line indicating the byte position in the file of that line (the program has been given itself as input):

```
0000:#include <stdlib.h>
0014:#include <stdio.h>
0027:
0028:void main()
0034:{
0036: char buffer[BUFSIZ];
004f: unsigned int count = 0, total = 0;
0076:
0077: while(gets(buffer) != NULL) {
0099: (void) printf("%04x:", total += count);
00c9: (void) printf("%s\n%n", buffer, (signed int *) &count);
0109: }
0113:
0114: exit(EXIT_SUCCESS);
012c:}
```

The %04x gives us a hexadecimal number in a 4 character wide field padded with 0's instead of blank spaces. There are two 'printf' statements since we don't want the size of the file position to be added to our running total. There are other ways to implement this without using '%n' since we can measure 'buffer' with 'strlen()' -- the '%n' specifier becomes more important when you can't measure the size of the item you are about to print (e.g. numbers).

Although most NeXT applications don't 'printf' to the TTY like traditional Unix programs, it's rare that you find a NeXT application that doesn't use 'printf' (in the form of 'sprintf') to format data strings or for debugging.

- Christopher